A framework of DYNAMIC data structures for string processing

Prezza, Nicola

Published in:
Leibniz International Proceedings in Informatics

Link to article, DOI:
10.4230/LIPIcs.SEA.2017.11

Publication date:
2017

Document Version
Publisher's PDF, also known as Version of record

A Framework of Dynamic Data Structures for String Processing*

Nicola Prezza

Technical University of Denmark, DTU Compute, Lyngby, Denmark
`npre@dtu.dk`

Abstract

In this paper we present `DYNAMIC`, an open-source C++ library implementing dynamic compressed data structures for string manipulation. Our framework includes useful tools such as searchable partial sums, succinct/gap-encoded bitvectors, and entropy/run-length compressed strings and FM indexes. We prove close-to-optimal theoretical bounds for the resources used by our structures, and show that our theoretical predictions are empirically tightly verified in practice. To conclude, we turn our attention to applications. We compare the performance of five recently-published compression algorithms implemented using `DYNAMIC` with those of state-of-the-art tools performing the same task. Our experiments show that algorithms making use of dynamic compressed data structures can be up to three orders of magnitude more space-efficient (albeit slower) than classical ones performing the same tasks.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases C++, dynamic, compression, data structure, bitvector, string

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.11

1 Introduction

Dynamism is an extremely useful feature in the field of data structures for string manipulation, and has been the subject of study in many recent works [5, 17, 24, 31, 20, 13]. These results showed that – in theory – it is possible to match information-theoretic upper and lower bounds on space of many problems related to dynamic data structures while still supporting queries in provably optimal time. From the practical point of view however, many of these results are based on complicated structures which prevent them from being competitive in practice. This is due to several factors that in practice play an important role but in theory are often poorly modeled, such as cache locality, branch prediction, disk accesses, context switches, memory fragmentation. Good implementations must take into account all these factors in order to be practical; this is the main reason why little work in this field has been done on the experimental side. An interesting and promising (but still under development) step in this direction is represented by `Memoria` [22], a C++14 framework providing general purpose dynamic data structures. Other libraries are also still under development (`ds-vector` [7]) or have been published but the code is not available [5, 17, 2]. Practical works considering weaker dynamic queries have also appeared. In [29] the authors consider rewritable arrays of integers (no indels or partial sums are supported). In [30] practical close-to-succinct dynamic tries are described (in this case, only navigational and child-append operations are supported). To the best of our knowledge, the only working implementation of a dynamic succinct bitvector

* Part of this work was done while the author was a PhD student at the University of Udine, Italy. Work supported by the Danish Research Council (DFF-4005-00267).

is [11]. This situation changes dramatically if the requirement of dynamism is dropped. In recent years, several excellent libraries implementing static data structures have been proposed: `sdsl` [12] (probably the most comprehensive, used, and tested), `pizza&chili` [25] (compressed indexes), `sux` [34], `succinct` [33], `libcds` [18]. These libraries proved that *static* succinct data structures can be very practical in addition to being theoretically appealing.

In view of this gap between theoretical and practical advances in the field, in this paper we present `DYNAMIC`: a C++11 library providing practical implementations of some basic succinct and compressed dynamic data structures for string manipulation: searchable partial sums, succinct/gap-encoded bitvectors, and entropy/run-length compressed strings and FM indexes. Our library has been extensively profiled and tested, and offers structures whose performance are provably close to the theoretical lower bounds (in particular, they approach succinctness and logarithmic queries). `DYNAMIC` is an open-source project and is available at [8].

We conclude by discussing the performance of five recently-published BWT/LZ77 compression algorithms [26, 28, 27] implemented with our library. On highly compressible datasets, our algorithms turn out to be up to three orders of magnitude more space-efficient than classical algorithms performing the same tasks.

2 The `DYNAMIC` library

The core of our library is a searchable partial sum with inserts data structure (SPSI in what follows). We start by formally defining the SPSI problem and showing how we solve it in `DYNAMIC`. We then proceed by describing how we use the SPSI structure as a building block to obtain the dynamic structures implemented in our library.

2.1 The Core: Searchable Partial Sums with Inserts

The Searchable Partial Sums With Inserts (SPSI) problem asks for a data structure PS maintaining a sequence $s_1, \ldots, s_m$ of non-negative integers and supporting the following operations on it:

- `PS.sum`$(i) = \sum_{j=1}^{i} s_j$;
- `PS.search`(x) is the smallest i such that $\sum_{j=1}^{i} s_j > x$;
- `PS.update`(i, δ): update s_i to $s_i + \delta$. δ can be negative as long as $s_i + \delta \geq 0$;
- `PS.insert`(i): insert 0 between s_{i-1} and s_i (if $i = 0$, insert in first position).

As discussed later, a consequence of the fact that our SPSI does not support `delete` operations is that also the structures we derive from it do not support `delete`; we plan to add this feature in our library in the future.

`DYNAMIC`'s SPSI is a B-tree storing integers $s_1, \ldots, s_m$ in its leaves and subtree size/partial sum counters in internal nodes. SPSI's operations are implemented by traversing the tree from the root to a target leaf and accessing internal nodes' counters to obtain the information needed for tree traversal. The choice of employing B-trees is motivated by the fact that a big node fanout translates to smaller tree height (w.r.t. a binary tree) and nodes that can fully fit in a cache line (i.e. higher cache efficiency). We use a leaf size l (i.e. number of integers stored in each leaf) always bounded by

$$0.5 \log m \leq l \leq \log m$$

and a node fanout $f \in \mathcal{O}(1)$. f should be chosen according to the cache line size; a bigger value for f reduces cache misses and tree height but increases the asymptotic cost of handling

single nodes. See Section 2.2 for a discussion on the maximum leaf size and f values used in practice in our implementation. Letting $l = c \cdot \log m$ be the size of a particular leaf, we call the coefficient $0.5 \leq c \leq 1$ the *leaf load*.

In order to improve space usage even further while still guaranteeing very fast operations, integers in the leaves are packed contiguously in word arrays and, inside each leaf $\mathcal{L}$, we assign to each integer the bit-size of the largest integer stored in $\mathcal{L}$. In the next section we prove that this simple blocking strategy leads to a space usage very close to the information-theoretic minimum number of bits needed to store the integers $s_1, \ldots, s_m$. It is worth to notice that – as opposed to other works such as [2] – inside each block we use a fixed-length integer encoding. Such an encoding allows much faster queries than variable-length integer codes (such as, e.g., Elias' delta or gamma) as in our strategy integers are stored explicitly and do not need to be decoded first. Whenever an integer overflows the maximum size associated to its leaf (after an `update` operation), we re-allocate space for all integers in the leaf. This operation takes $\mathcal{O}(\log m)$ time, so it does not asymptotically increase the cost of `update` operations. Crucially, in each leaf we allocate space only for the integers actually stored inside it, and re-allocate space for the whole leaf whenever we insert a new integer or we split the leaf. With this strategy, we do not waste space for half-full leaves[1]. Note, moreover, that since the size of each leaf is bounded by $\Theta(\log m)$, re-allocating space for the whole leaf at each insertion does not asymptotically slow down `insert` operations.

2.1.1 Theoretical Guarantees

Let us denote with $m/\log m \leq L \leq 2m/\log m$ the total number of leaves, with $\mathcal{L}_j$, $0 \leq j < L$, the j-th leaf of the B-tree (using any leaf order), and with $I \in \mathcal{L}_j$ an integer belonging to the j-th leaf. The total number of bits stored in the leaves of the tree is

$$\sum_{0 \leq j < L} \sum_{I \in \mathcal{L}_j} max\_bitsize(\mathcal{L}_j)$$

where $max\_bitsize(\mathcal{L}_j) = \max_{I \in \mathcal{L}_j} (bitsize(I))$ is the bit-size of the largest $I \in \mathcal{L}_j$, and $bitsize(x)$ is the number of bits required to write number x in binary: $bitsize(0) = 1$ and $bitsize(x) = \lfloor \log_2 x \rfloor + 1$, for $x > 0$. The above quantity is equal to

$$\sum_{0 \leq j < L} c_j \cdot \log m \cdot max\_bitsize(\mathcal{L}_j)$$

where $0.5 \leq c_j \leq 1$ is the j-th leaf load. Since leaves' loads are always upper-bounded by 1, the above quantity is upper-bounded by

$$\log m \sum_{0 \leq j < L} max\_bitsize(\mathcal{L}_j)$$

which, in turn, is upper-bounded by

$$\log m \sum_{0 \leq j < L} bitsize\left(\sum_{I \in \mathcal{L}_j} I\right) \leq \log m \sum_{0 \leq j < L} 1 + \log_2 \left(1 + \sum_{I \in \mathcal{L}_j} I\right).$$

In the above inequality, we use the upper-bound $bitsize(x) \leq 1 + \log_2(1 + x)$ to deal with the case $x = 0$. Let $M = m + \sum_{i=1}^{m} s_i = m + \sum_{0 \leq j < L} \sum_{I \in \mathcal{L}_j} I$ be the sum of all integers

[1] in practice, to speed up operations we allow a small fraction of the leaf to be empty.

stored in the structure plus m. From the concavity of log and from $L \leq 2m/\log m$, it can be derived that the above quantity is upper-bounded by

$$2m \cdot (\log(M/m) + \log\log m + 1) \ .$$

To conclude, we store $\mathcal{O}(1)$ pointers/counters of $\mathcal{O}(\log M)$ bits each per leaf and internal node. We obtain:

▶ **Theorem 1.** *Let $s_1, \ldots, s_m$ be a sequence of m non-negative integers and $M = m + \sum_{i=1}^{m} s_i$. The partial sum data structure implemented in* `DYNAMIC` *takes at most*

$$2 \cdot m\,(\log(M/m) + \log\log m + \mathcal{O}(\log M/\log m))$$

bits of space and supports `sum`*,* `search`*,* `update`*, and* `insert` *operations on the sequence $s_1, \ldots, s_m$ in $\mathcal{O}(\log m)$ time.*

Our implementation uses the standard C++ memory allocator to allocate memory for the growing dynamic structures. As shown in the experimental section, this choice results in a non-negligible fraction of memory being wasted due to memory fragmentation. Ad-hoc allocators such as the one discussed in [5] can significantly alleviate this effect. In our experiments we observed that – even taking into account memory fragmentation – the bit-size of our dynamic partial sum structure is well approximated by function $1.19 \cdot m\,(\log(M/m) + \log\log m + \log M/\log m)$. See the experimental section for full details.

2.2 Plug and Play with Dynamic Structures

The SPSI structure described in the previous section is used as a building block to obtain all dynamic structures of our library. In `DYNAMIC`, the SPSI structure's type name is `spsi` and is parametrized on three template arguments: the leaf type (here, the type `packed_vector` is always used[2]), the leaf size and the node fanout. `DYNAMIC` defines two SPSI types with two different combinations of these parameters:

```
typedef spsi<packed_vector,256,16> packed_spsi;
typedef spsi<packed_vector,8192,16> succinct_spsi;
```

The reasons for the particular values chosen for the leaf size and node fanout will be discussed later. We use these two types as basic components in the definition our structures.

2.2.1 Gap-Encoded Bitvectors

`DYNAMIC` implements gap-encoded bitvectors using an SPSI to encode gap lengths: bitvector $0^{s_1-1}10^{s_2-1}1\ldots0^{s_m-1}1$ ($s_i > 0$) is encoded with a partial sum on the sequence $s_1, \ldots, s_m$. For space reasons, we do not describe how to reduce the gap-encoded bitvector problem to the SPSI problem; the main idea is to reduce bitvector's `access` and `rank` operations to SPSI's `search` operations, bitvector's `select` operations to SPSI's `sum` operations, bitvector's $\texttt{insert}_1$ operations to SPSI's `insert` operations, and bitvector's $\texttt{insert}_0/\texttt{delete}_0$ operations to SPSI's `update` operations.

`DYNAMIC`'s name for the dynamic gap-encoded bitvector class is `gap_bitvector`. The class is a template on the SPSI type. We plug `packed_spsi` in `gap_bitvector` as follows:

[2] `packed_vector` is simply a packed vector of fixed-size integers supporting all SPSI operations in linear time.

```
typedef gap_bitvector<packed_spsi> gap_bv;
```

and obtain:

▶ **Theorem 2.** *Let $B \in \{0,1\}^n$ be a bit-sequence with b bits set. The dynamic gap-encoded bitvector* `gap_bv` *implemented in* `DYNAMIC` *takes at most*

$$2 \cdot b\,(\log(n/b) + \log\log b + \mathcal{O}(\log n/\log b))\,(1 + o(1))$$

bits of space and supports `rank`, `select`, `access`, `insert`, *and* $\texttt{delete}_0$ *operations on B in $\mathcal{O}(\log b)$ time.*

In our experiments, the optimal node fanout for the SPSI structure employed in this component turned out to be 16, while the optimal leaf size 256 (these values represented a good compromise between query times and space usage). Our benchmarks show (see the experimental section for full details) that the bit-size of our dynamic gap-encoded bitvector is well approximated by function $1.19 \cdot b\,(\log(n/b) + \log\log b + \log n/\log b)$.

It is worth to notice that an alternative efficient implementation of bitvectors is run-length encoding (RLE): a bitvector $0^{k_1}1^{k_2}0^{k_3}\ldots$ can be represented with an SPSI on the integer sequence $k_1, k_2, k_3, \ldots$. This representation results advantageous in cases where the underlying bitvector contains also long runs of bits set (e.g. a wavelet tree on a string with long equal-letter runs). We preferred using gap-encoding for two main reasons. First of all, in our library gap-encoded bitvectors are at the core of run-length encoded strings (more details in Section 2.2.3). In such structures, every equal-letter run is marked with a bit set in a gap-encoded bitvector. This breaks the symmetry between zeros and ones in the bitvector as strings with long equal-letter runs will generate bitvectors with very few bits set. Then, note that the above-mentioned RLE bitvector representation allows for efficient `access` operations, but does not support (fast) `rank`. To support `rank`, one should store separately the cumulated lengths of runs of zeros (or ones). This is exactly what our run-length compressed string (on the alphabet $\{0,1\}$) does (see Section 2.2.3).

2.2.2 Succinct Bitvectors and Entropy-Compressed Strings

Let n be the bitvector length. Dynamic succinct bitvectors can be implemented using an SPSI where all $m = n$ stored integers are either 0 or 1. At this point, `rank` operations on the bitvector correspond to `sum` on the partial sum structure, and `select` operations on the bitvector can be implemented with `search` on the partial sum structure[3]. `access` and `insert` operations on the bitvector correspond to exactly the same operations on the partial sum structure. Note that in this case we can accelerate operations in the leaves by a factor of $\log n$ by using constant-time built-in bitwise operations such as `popcount`, masks and shifts. This allows us to use bigger leaves containing $\Theta(\log^2 n)$ bits, which results in a total number of internal nodes bounded by $\mathcal{O}(n/\log^2 n)$. The overhead for storing internal nodes is therefore of $o(n)$ bits. Moreover, since in the leaves we allocate only the *necessary* space to store the bitvector's content (i.e. we do not allow empty space in the leaves), it easily follows that the dynamic bitvector structure implemented in `DYNAMIC` takes $n + o(n)$ bits of space and supports all operations in $\mathcal{O}(\log n)$ time.

[3] Actually, `search` permits to implement only $\texttt{select}_1$. $\texttt{select}_0$ can however be easily simulated with the same solution used for `search` by replacing each integer $x \in \{0,1\}$ with $1 - x$ at run time. This solution does not increase space usage.

In our experiments, the optimal node fanout for the SPSI stucture employed in the succinct bitvector structure turned out to be 16, while the optimal leaf size 8192. `DYNAMIC`'s name for the dynamic succinct bitvector is `succinct_bitvector`. The class is a template on the SPSI type. `DYNAMIC` defines its dynamic succinct bitvector type as:

```
typedef succinct_bitvector<succinct_spsi> suc_bv;
```

We obtain:

▶ **Theorem 3.** *Let $B \in \{0,1\}^n$ be a bit-sequence. The dynamic succinct bitvector data structure* ***suc_bv*** *implemented in* ***DYNAMIC*** *takes $n + o(n)$ bits of space and supports* ***rank****,* ***select****,* ***access****, and* ***insert*** *operations on B in $\mathcal{O}(\log n)$ time.*

In our experiments (see the experimental section) the size of our dynamic succinct bitvector was always upper-bounded by $1.23 \cdot n$ bits. The 23% overhead on top of the optimal size comes mostly from memory fragmentation (16%). The remaining 7% comes from succinct structures on top of the bit-sequence.

Dynamic compressed strings are implemented with a wavelet tree built upon dynamic succinct bitvectors. We explicitly store the topology of the tree ($\mathcal{O}(|\Sigma| \log n)$ bits) instead of encoding it implicitly in a single bitvector. This choice is space-inefficient for very large alphabets, but reduces the number of `rank/select` operations on the bitvector(s) with respect of a wavelet tree stored as a single bitvector. `DYNAMIC`'s compressed strings (wavelet trees) are a template on the bitvector type. `DYNAMIC` defines its dynamic string type as:

```
typedef wt_string<suc_bv> wt_str;
```

The user can choose at construction time whether to use a Huffman, fixed-size, or Gamma encoding for the alphabet. Gamma encoding is useful when the alphabet size is unknown at construction time. The Huffman encoding of the string uses at most $n(H_0 + 1)$ bits; a Huffman-shaped wavelet tree only adds a low-order overhead on top of this representation. In our library, we store the Huffman tree topology using pointers (instead of concatenating the wavelet tree's bitvectors into a single bitvector). This strategy reduces the number of operations needed to navigate the tree, but adds a $\mathcal{O}(|\Sigma| \log n)$-bits overhead. We obtain:

▶ **Theorem 4.** *Let $S \in \Sigma^n$ be a string with zero-order entropy equal to H_0. The Huffman-compressed dynamic string data structure* ***wt_str*** *implemented in* ***DYNAMIC*** *takes*

$$n(H_0 + 1)(1 + o(1)) + \mathcal{O}(|\Sigma| \log n)$$

bits of space and supports ***rank****,* ***select****,* ***access****, and* ***insert*** *operations on S in average $\mathcal{O}((H_0 + 1) \log n)$ time.*

When a fixed-size encoding is used (i.e. $\lceil \log_2 |\Sigma| \rceil$ bits per character), the structure takes $n \log |\Sigma|(1 + o(1)) + \mathcal{O}(|\Sigma| \log n)$ bits of space and supports all operations in $\mathcal{O}(\log |\Sigma| \cdot \log n)$ time.

2.2.3 Run-Length Encoded Strings

To run-length encode a string $S \in \Sigma^n$, we adopt the approach described in [32]. We store one character per run in a string $H \in \Sigma^r$, we mark the end of the runs with a bit set in a bit-vector $V_{all}[0, \ldots, n-1]$, and for every $c \in \Sigma$ we store all c-runs lengths consecutively in a bit-vector V_c as follows: every m-length c-run is represented in V_c as $0^{m-1}1$.

▶ **Example 5.** Let $S = bc\#bbbbccccbaaaaaaaaaaa$. We have: $H = bc\#bcba$, $V_{all} = 11100010001100000000001$, $V_a = 00000000001$, $V_b = 100011$, $V_c = 10001$, and $V_\# = 1$

By encoding H with a wavelet tree and gap-compressing all bitvectors, we achieve run-length compression. It can be easily shown that this representation allows supporting `rank`, `select`, `access`, and `insert` operations on S, but for space reasons we do not give these details here. In `DYNAMIC`, the run-length compressed string type `rle_string` is a template on the gap-encoded bitvector type (bitvectors V_{all} and V_c, $c \in \Sigma$) and on the dynamic string type (run heads H). We plug the structures of the previous sections in the above representation as follows:

```
typedef rle_string<gap_bv, wt_str> rle_str;
```

and obtain:

▶ **Theorem 6.** *Let $S \in \Sigma^n$ be a string with r_S equal-letter runs. The dynamic run-length encoded string data structure* ***`rle_str`*** *implemented in* ***`DYNAMIC`*** *takes*

$$r_S \cdot (4\log(n/r_S) + \log|\Sigma| + 4\log\log r_S + \mathcal{O}(\log n/\log r_S))\,(1 + o(1)) + \mathcal{O}(|\Sigma|\log n)$$

bits of space and supports ***`rank`****,* ***`select`****,* ***`access`****, and* ***`insert`*** *operations on S in $\mathcal{O}(\log|\Sigma| \cdot \log r_S)$ time.*

2.2.4 Dynamic FM Indexes

An FM index [10] is a data structure supporting `rank` operations over the Burrows-Wheeler transform [3] (BWT) of the text, plus a suitable sampling mechanism that associates text positions to a subset of BWT positions (i.e. a suffix array sampling). Such a data structure takes space close to that of the compressed text (provided that the string structure used is compressed) and supports fast counting and locating occurrences of a pattern in the text. If the data structure used to represent the string supports also `insert` operations, then the FM index support also left-extensions of the text [4, 21, 20].

We obtain dynamic FM indexes by combining a dynamic Burrows-Wheeler transform with a sparse dynamic vector storing the suffix array sampling. In `DYNAMIC`, the BWT is a template class parametrized on the L-column and F-column types. For the F column, a run-length encoded string is always used. `DYNAMIC` defines two types of dynamic Burrows-Wheeler transform structures (wavelet-tree/run-length encoded):

```
typedef bwt<wt_str,rle_str> wt_bwt;
typedef bwt<rle_str,rle_str> rle_bwt;
```

Dynamic sparse vectors are implemented inside the FM index class using a dynamic bitvector marking sampled BWT positions and a dynamic sequence of integers (an SPSI) storing non-null values. We combine a Huffman-compressed BWT with a succinct bitvector and an SPSI:

```
typedef fm_index<wt_bwt, suc_bv, packed_spsi> wt_fmi;
```

and obtain:

▶ **Theorem 7.** *Let $S \in \Sigma^n$ be a string with zero-order entropy equal to H_0, $P \in \Sigma^m$ a pattern occurring occ times in T, and k the suffix array sampling rate. The dynamic Huffman-compressed FM index* ***`wt_fmi`*** *implemented in* ***`DYNAMIC`*** *takes*

$$n(H_0 + 2)(1 + o(1)) + \mathcal{O}(|\Sigma|\log n) + (n/k)\log n$$

bits of space and supports:

- *`access` to BWT characters in average* $\mathcal{O}((H_0+1)\log n)$ *time*
- *`count` in average* $\mathcal{O}(m(H_0+1)\log n)$ *time*
- *`locate` in average* $\mathcal{O}((m+occ\cdot k)(H_0+1)\log n)$ *time*
- *text `left-extension` in average* $\mathcal{O}((H_0+1)\log n)$ *time*

If a plain alphabet encoding is used, all (H_0+1) *terms are replaced by* $\log|\Sigma|$ *and times become worst-case.*

If, instead, we combine a run-length compressed BWT with a gap-encoded bitvector and an SPSI as follows:

```
typedef fm_index<rle_bwt, gap_bv, packed_spsi> rle_fmi;
```

we obtain:

▶ **Theorem 8.** *Let* $S\in\Sigma^n$ *be a string whose BWT has* r *runs,* $P\in\Sigma^m$ *a pattern occurring* occ *times in* T*, and* k *the suffix array sampling rate. The dynamic run-length compressed FM index `rle_fmi` implemented in `DYNAMIC` takes*

$$r\cdot(4\log(n/r)+\log|\Sigma|+4\log\log r+\mathcal{O}(\log n/\log r))\,(1+o(1))+\mathcal{O}(|\Sigma|\log n)+(n/k)\log n$$

bits of space and supports:

- *`access` to BWT characters in* $\mathcal{O}(\log|\Sigma|\cdot\log r)$ *time*
- *`count` in* $\mathcal{O}(m\cdot\log|\Sigma|\cdot\log r)$ *time*
- *`locate` in* $\mathcal{O}((m+occ\cdot k)(\log|\Sigma|\cdot\log r))$ *time*
- *text `left-extension` in* $\mathcal{O}(\log|\Sigma|\cdot\log r)$ *time*

The suffix array sample rate k can be chosen at construction time.

3 Experimental Evaluation

We start by presenting detailed benchmarks of our gap-encoded and succinct bitvectors, that are at the core of all other library's structures. We then turn our attention to applications: we compare the performance of five recently-published compression algorithms implemented with `DYNAMIC` against those of state-of-the-art tools performing the same tasks and working in uncompressed space. All experiments were performed on a `intel core i7` machine with 12 GB of RAM running Linux Ubuntu 16.04.

3.1 Benchmarks: Succinct and Gap-Encoded Bitvectors

We built 34 gap-encoded (`gap_bv`) and 34 succinct (`suc_bv`) bitvectors of length $n=500\cdot10^6$ bits, varying the frequency b/n of bits set in the interval $[0.0001, 0.99]$. In each experiment, we first built the bitvector by performing n $\texttt{insert}_b$ queries, b being equal to 1 with probability b/n, at uniform random positions. After building the bitvector, we executed n $\texttt{rank}_0$, n $\texttt{rank}_1$, n $\texttt{select}_0$, n $\texttt{select}_1$, and n `access` queries at uniform random positions. Running times of each query were averaged over the n repetitions. We measured memory usage in two ways: (i) internally by counting the total number of bits allocated by our procedures – this value is denoted as *allocated* memory in our plots –, and (ii) externally using the tool `/usr/bin/time` – this value is denoted as *RSS* in our plots (Resident Set Size).

Working space. We fitted measured RSS memory with the theoretical predictions of Section 2.1.1 using a linear regression model. Parameters of the model were inferred using the statistical tool `R` (function `lm`). In detail, we fitted RSS memory in the range $b/n \in [0, 0.1]$[4] with function $k \cdot f(n, b) + c$, where: $f(n, b) = b \cdot (\log(n/b) + \log\log b + \log n/\log b)$ is our theoretical prediction (recall that memory occupancy of our gap-encoded bitvector should never exceed $2f(n, b)$), k is a scaling factor accounting for memory fragmentation and average load distribution in the B-tree, and c is a constant accounting for the weight of loaded C++ libraries (this component cannot be excluded from the measurements of the tool `/usr/bin/time`). Function `lm` provided us with parameters $k = 1.19$ and $c = 28,758,196$ bits $\approx 3.4MB$. The value for c was consistent with the space measured with b/n close to 0.

Figures 1 and 2 show memory occupancy of `DYNAMIC`'s bitvectors as a function of the frequency b/n of bits set. In Figure 1 we compare both bitvectors. In Figure 2 we focus on the behavior of our gap-encoded bitvector in the interval $b/n \in [0, 0.1]$. In these plots we moreover show the growth of function $1.19 \cdot f(n, b) + 28,758,196$. Plot in Figure 1 shows that our theoretical prediction fits almost perfectly the memory usage of our gap-encoded bitvector for $b/n \leq 0.7$. The plot suggests moreover that for $b/n \geq 0.1$ it is preferable to use our succinct bitvector rather than the gap-encoded one. As far as the gap-encoded bitvector is concerned, memory fragmentation[5] amounts to approximately 15% of the allocated memory for $b/n \leq 0.5$. This fraction increases to 24% for b/n close to 1. We note that RSS memory usage of our succinct bitvector never exceeds $1.29n$ bits: the overhead of $0.29n$ bits is distributed among (1) `rank/select` succinct structures ($\approx 0.07n$ bits) (2) loaded C++ libraries (a constant amounting to approximately 3.4 MB, i.e. $\approx 0.06n$ bits in this case), and memory fragmentation ($\approx 0.16n$ bits). Excluding the size of C++ libraries (which is constant), our bitvector's size never exceeds $1.23n$ bits (being $1.20n$ bits on average).

Query times. Plots in Figures 3-6 show running times of our bitvectors on all except $\texttt{rank}_0$ and $\texttt{select}_0$ queries (which were very close to those of $\texttt{rank}_1$ and $\texttt{select}_1$ queries, respectively). We used a linear regression model (inferred using `R`'s function `lm`) to fit query times of our gap-encoded bitvector with function $c + k \cdot \log b$. Query times of our succinct bitvector were interpolated with a constant (with n fixed). These plots show interesting results. First of all, our succinct bitvector supports extremely fast ($0.01\mu s$ on average) `access` queries. `rank` and `select` queries are, on average, 15 times slower than `access` queries. As expected, `insert` queries are very slow, requiring – on average – 390 times the time of `access` queries and 26 times that of `rank/select` queries. On all except `access` queries, running times of our gap-encoded bitvector are faster than (or comparable to) those of our succinct bitvector for $b/n \leq 0.1$. Combined with the results depicted in Plot 1, these considerations confirm that for $b/n \leq 0.1$ our gap-encoded bitvector should be preferred to the succinct one. `access`, `rank`, and `select` queries are all supported in comparable times on our gap-encoded bitvector ($\approx 0.05 \cdot \log b\ \mu s$), and are one order of magnitude faster than `insert` queries.

[4] For $b/n \geq 0.1$ it becomes more convenient – see below – to use our succinct bitvector, so we considered it more useful to fit memory usage in $b \in [0, 0.1]$. In any case – see plot 1 – the inferred model fits the experimental data well in the (wider) interval $b/n \in [0, 0.7]$.

[5] We estimated the impact of memory fragmentation by comparing RSS and allocated memory, after subtracting from RSS the estimated weight – approximately 3.4 MB – of loaded C++ libraries.

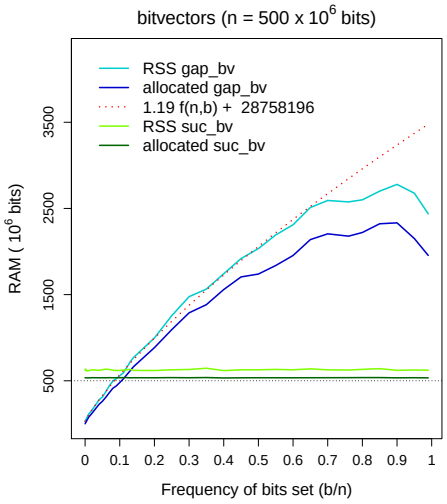

Figure 1 Memory occupancy of `DYNAMIC`'s bitvectors. We show the growth of function $f(n, b) = b(\log(n/b) + \log\log b + \log n/\log b)$ opportunely scaled to take into account memory fragmentation and the weight of loaded C++ libraries.

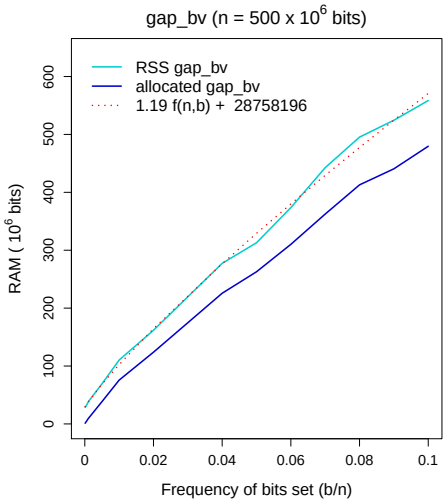

Figure 2 Memory occupancy of `DYNAMIC`'s gap-encoded bitvector in the interval $b/n \in [0, 0.1]$ (for $b/n > 0.1$ the succinct bitvector is more space-efficient then the gap-encoded one). The picture shows that allocated memory closely follows our theoretical prediction (function $f(n, b)$).

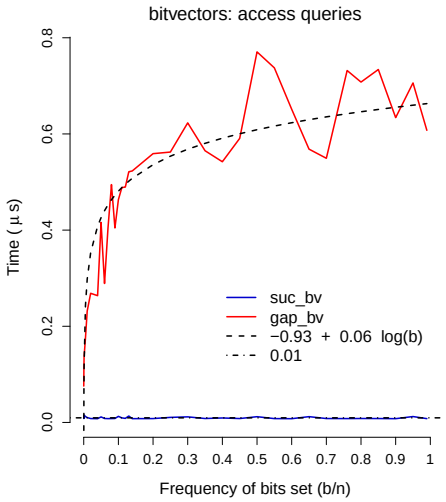

Figure 3 Running times of our bitvectors on `access` queries. Bitvectors' size is $n = 5 \times 10^8$ bits.

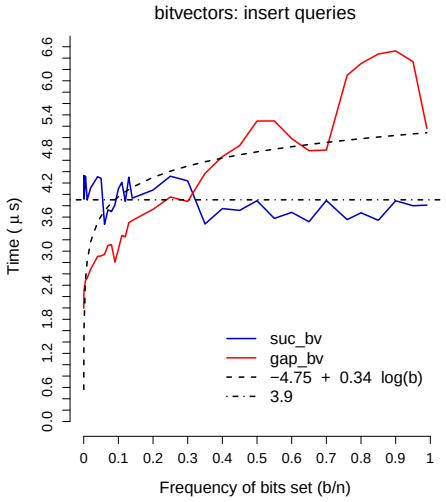

Figure 4 Running times of our bitvectors on `insert` queries. Bitvectors' size is $n = 5 \times 10^8$ bits.

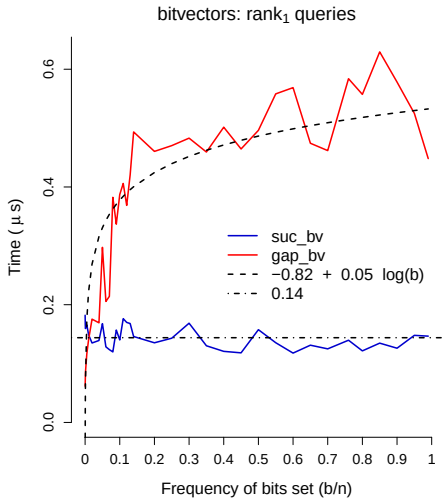

Figure 5 Running times of our bitvectors on $\texttt{rank}_1$ queries. Bitvectors' size is $n = 5 \times 10^8$ bits.

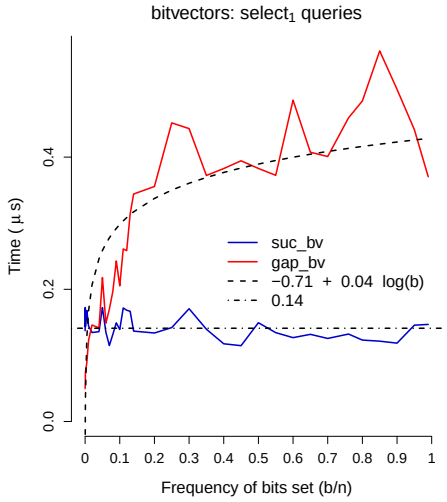

Figure 6 Running times of our bitvectors on $\texttt{select}_1$ queries.

3.2 An Application: Space-Efficient Compression Algorithms

We used `DYNAMIC` to implement five recently-published algorithms [26, 28, 27] computing the Burrows-Wheeler transform [3] (BWT) and the Lempel-Ziv 77 factorization [35] (LZ77) within compressed working space: `cw-bwt` [27] builds a BWT within $n(H_k + 1) + o(n \log \sigma)$ bits of working space by breaking it in contexts and encoding each context with a zero-order compressed string; `rle-bwt` builds the BWT within $\Theta(r)$ words of working space using the structure of Theorem 6; `h0-lz77` [28] computes LZ77 online within $n(H_0 + 2) + o(n \log \sigma)$ bits using a dynamic zero-order compressed FM index; `rle-lz77-1` and `rle-lz77-2` [26] build LZ77 within $\Theta(r)$ words of space by employing a run-length encoded BWT augmented with a suffix array sampling based on BWT equal-letter runs and LZ77 factors, respectively. Implementations of these algorithms can be found within the `DYNAMIC` library [8]. We compared running times and working space of our algorithms against those of less space-efficient (but faster) state-of-the-art tools solving the same problems. BWT construction tools: `se-sais` [1, 12] ($\Theta(n)$ Bytes of working space), `divsufsort` [23, 12] ($\Theta(n)$ words), `bwte` [9] (constant user-defined working space; we always used 256 MB), `dbwt` [6] ($\Theta(n)$ Bytes). LZ77 factorization tools: `isa6r` [16, 19] ($\Theta(n)$ words), `kkp1s` [15, 19] ($\Theta(n)$ words), `lzscan` [14, 19] ($\Theta(n)$ Bytes). We generated two highly repetitive text collections by downloading all versions of the *Boost* library (`github.com/boostorg/boost`) and all versions of the English *Einstein*'s Wikipedia page (`en.wikipedia.org/wiki/Albert_Einstein`). Both datasets were truncated to $5 \cdot 10^8$ Bytes to limit RAM usage of the and computation times of the tested tools. The sizes of the 7-Zip-compressed datasets (`www.7-zip.org`) were 120 KB (Boost) and 810 KB (Einstein). The datasets can be found within the `DYNAMIC` library [8] (folder `/datasets/`). RAM usage and running times of the tools were measured using the executable `/usr/bin/time`.

In Figure 7 we report our results. Solid and a dashed horizontal lines show the datasets' sizes before and after compression with 7-Zip, respectively. Our tools are highlighted in red. We can infer some general trends from the plots. Our tools use always less space than the plain text, and from one to three orders of magnitude more space than the 7-Zip-compressed text. `h0-lz77` and `cw-bwt` (entropy compression) always have working space very close to (and always smaller than) the plain text, with `cw-bwt` (k-th order compression) being more space-efficient than `h0-lz77` (0-order compression). On the other hand, tools using a run-length compressed BWT – `rle-bwt`, `rle-lz77-1`, and `rle-lz77-2` – are up to two orders of magnitude more space-efficient than `h0-lz77` and `cw-bwt` in most of the cases. This is a consequence of the fact that run-length encoding of the BWT is particularly effective in compressing repetitive datasets. `bwte` represents a good trade-off in both running times and working space between tools working in compressed and uncompressed working space. `kkp1s` is the fastest tool, but uses a working space that is one order of magnitude larger than the uncompressed text and three orders of magnitude larger than that of `rle-bwt`, `rle-lz77-1`, and `rle-lz77-2`. As predicted by theory, tools working in compact working space (`lzscan`, `se-sais`, `dbwt`) use always slightly more space than the uncompressed text, and one order of magnitude less space than tools working in $\mathcal{O}(n)$ words. To conclude, the plots show that the price to pay for using complex dynamic data structures is high running times: our tools are up to three orders of magnitude slower than tools working in $\Theta(n)$ words of space. This is mainly due to the large number of `insert` operations – one per text character – performed by our algorithms to build the dynamic FM indexes.

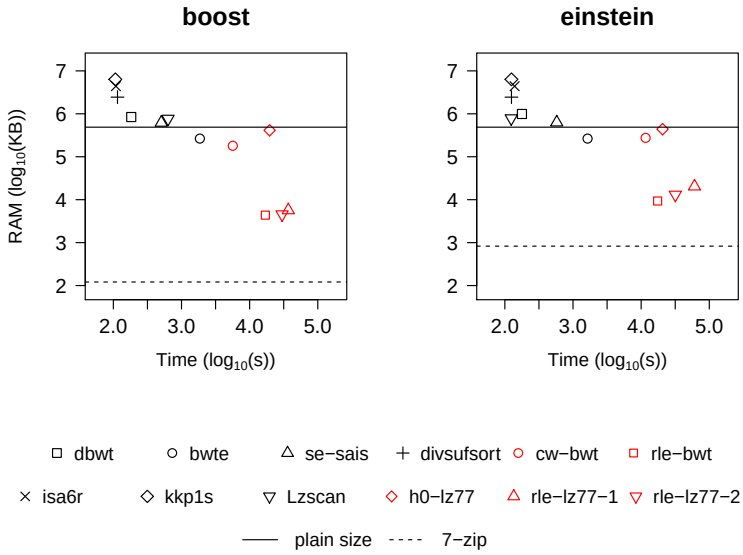

Figure 7 BWT and LZ77 compression algorithms. In red: tools implemented using `DYNAMIC`. Solid/dashed lines: space of the input files before and after 7-Zip compression, respectively.

References

1 Timo Beller, Maike Zwerger, Simon Gog, and Enno Ohlebusch. Space-Efficient Construction of the Burrows-Wheeler Transform. In *String Processing and Information Retrieval*, pages 5–16. Springer, 2013.

2 Daniel K. Blandford and Guy E. Blelloch. Compact representations of ordered sets. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 11–19. Society for Industrial and Applied Mathematics, 2004.

3 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm, 1994.

4 Ho-Leung Chan, Wing-Kai Hon, Tak-Wah Lam, and Kunihiko Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms (TALG)*, 3(2):21, 2007.

5 Joshimar Cordova and Gonzalo Navarro. Practical dynamic entropy-compressed bitvectors with applications. In *International Symposium on Experimental Algorithms*, pages 105–117. Springer, 2016.

6 dbwt: direct construction of the BWT. `http://researchmap.jp/muuw41s7s-1587/#_1587`. Accessed: 2016-11-17.

7 ds-vector: C++ library for dynamic succinct vector. `https://code.google.com/archive/p/ds-vector/`. Accessed: 2016-11-17.

8 DYNAMIC: dynamic succinct/compressed data structures library. `https://github.com/xxsds/DYNAMIC`. Accessed: 2017-01-22.

9 Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012.

10 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

11 bitvector: succinct dynamic bitvector implementation. `https://github.com/nicola-gigante/bitvector`. Accessed: 2016-11-17.

12 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014. `doi:10.1007/978-3-319-07959-2_28`.

13 Roberto Grossi, Rajeev Raman, Satti Srinivasa Rao, and Rossano Venturini. Dynamic compressed strings with random access. In *International Colloquium on Automata, Languages, and Programming*, pages 504–515. Springer, 2013.

14 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lightweight Lempel-Ziv parsing. In *Experimental Algorithms*, pages 139–150. Springer, 2013.

15 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Combinatorial Pattern Matching*. Springer, 2013.

16 Dominik Kempa and Simon J. Puglisi. Lempel-Ziv factorization: Simple, fast, practical. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 103–112. Society for Industrial and Applied Mathematics, 2013.

17 Patrick Klitzke and Patrick K. Nicholson. A general framework for dynamic succinct and compressed data structures. *Proceedings of the 18th ALENEX*, pages 160–173, 2016.

18 libcds: compact data structures library. `https://github.com/fclaude/libcds`. Accessed: 2016-11-17.

19 LZ77 factorization algorithms. `https://www.cs.helsinki.fi/group/pads/lz77.html`. Accessed: 2016-05-20.

20 Veli Mäkinen and Gonzalo Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 4(3):32, 2008.

21 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. of Computational Biology*, 17(3):281–308, 2010.

22 Memoria: C++14 framework providing general purpose dynamic data structures. `https://bitbucket.org/vsmirnov/memoria/wiki/Home`. Accessed: 2016-11-17.

23 Y. Mori. Short description of improved two-stage suffix sorting algorithm, 2005.

24 Gonzalo Navarro and Yakov Nekrich. Optimal dynamic sequence representations. *SIAM Journal on Computing*, 43(5):1781–1806, 2014.

25 Pizza&Chili corpus. `http://pizzachili.dcc.uchile.cl`. Accessed: 2016-07-25.

26 A. Policriti and N. Prezza. Computing LZ77 in Run-Compressed Space. In *2016 Data Compression Conference (DCC)*, pages 23–32, March 2016. `doi:10.1109/DCC.2016.30`.

27 Alberto Policriti, Nicola Gigante, and Nicola Prezza. Average linear time and compressed space construction of the Burrows-Wheeler transform. In *International Conference on Language and Automata Theory and Applications*, pages 587–598. Springer, 2015.

28 Alberto Policriti and Nicola Prezza. Fast online Lempel-Ziv factorization in compressed space. In *International Symposium on String Processing and Information Retrieval*, pages 13–20. Springer, 2015.

29 Andreas Poyias, Simon J Puglisi, and Rajeev Raman. Compact Dynamic Rewritable (CDRW) Arrays. In *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 109–119. SIAM, 2017.

30 Andreas Poyias and Rajeev Raman. Improved practical compact dynamic tries. In *International Symposium on String Processing and Information Retrieval*, pages 324–336. Springer, 2015.

31 Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct dynamic data structures. In *Workshop on Algorithms and Data Structures*, pages 426–437. Springer, 2001.

32 Jouni Sirén, Niko Välimäki, Veli Mäkinen, and Gonzalo Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *String Processing and Information Retrieval*, pages 164–175. Springer, 2009.

33 succinct library. `https://github.com/ot/succinct`. Accessed: 2016-11-17.

34 sux library. `http://sux.di.unimi.it/`. Accessed: 2016-11-17.

35 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.